

FlashSim: A Simulator for NAND Flash-based Solid-State Drives

Youngjae Kim[†] Brendan Tauras[‡] Aayush Gupta[†] Bhuvan Urgaonkar[†]

Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802

[†]{youkim, axg354, bhuvan}@cse.psu.edu, [‡]{tauras}@psu.edu

Abstract

NAND Flash memory-based Solid-State Disks (SSDs) are becoming popular as the storage media in domains ranging from mobile laptops to enterprise-scale storage systems due to a number of benefits (e.g., lighter weights, faster access times, lower power consumption, higher resistance to vibrations) they offer over the conventionally popular Hard Disk Drives (HDDs). While a number of well-regarded simulation environments exist for HDDs, the same is not yet true for SSDs. This is due to SSDs having been in the storage market for relatively less time as well as the lack of information (hardware configuration and software methods) about state-of-the-art SSDs that is publicly available. We describe the design and implementation of FlashSim, a simulator aimed at filling this void in performance evaluation of emerging storage systems that employ SSDs. FlashSim is an event-driven simulator that follows the object-oriented programming paradigm for modularity. We have validated the performance of FlashSim against a number of commercial SSDs for behavioral similarity. We have also used FlashSim to compare the performance of SSD devices employing different Flash Translation Layer (FTL) schemes, and analyzed the energy consumption of different FTL schemes in the SSD. FlashSim has been written to be inter-operable with the well-regarded DiskSim simulator, thus enabling the simulation of a variety of “hybrid” storage systems employing combinations of SSDs and HDDs. Given the current interest in such hybrid systems as opposed to systems with SSDs replacing HDDs (due to higher price), we believe this to be an especially useful feature of FlashSim.

1. Introduction

Recently, NAND Flash memory has become the main storage media for embedded devices, such as PDAs and music players. NAND Flash memory is now also being used in systems ranging from laptop and desktop computers to enterprise-scale storage servers. NAND Flash memory offers a number of benefits over the conventional hard disk drives (HDDs). These benefits include lower power consumption, lighter weight, higher resilience to external shock, the ability

to sustain hotter operating regimes, and faster access times (with some exceptions that arise due to random writes). Unlike HDDs, NAND flash memory based Solid-State Disks (SSDs) have no mechanical moving parts, such as a spindle and voice-coil motors. Despite these benefits, a storage system designer needs to carefully consider the use of SSDs because they also have some notable weaknesses. The main weaknesses of SSDs include a higher price (\$/GB) than HDDs, writes being 4-5 times slower than reads, slowdown in device throughput during periods of garbage collection that are hastened by small, random writes [14], and limited lifetime (10K-1M erase cycles per block) [4].

In order to overcome the limitations described above, a variety of complementary approaches have been proposed. For example, Multi-level Cell (MLC) technology gives higher density and cost per GB than Single-level Cell (SLC) [19]. The downside of MLC is that read and write times of MLC are slower. Consequently, there are current attempts to employ combinations of SLC and MLC Flash chips in SSDs. Numerous techniques for efficient address translation, garbage collection, and wear-leveling in the Flash Translation Layer (FTL) (more details in Section 2) have been explored to improve the performance of the SSD devices and/or providing longer lifetimes.

The design and implementation of cost efficient, reliable SSDs requires faithful and accurate evaluation test-beds for evaluating new algorithms for specific software components (such as those that constitute the FTL) within different hardware configurations of the SSD before implementing them in the actual firmware. The fact that significant aspects of the techniques employed within SSDs are unknown to the public due to technology property issues further adds to the urgency of having such a test-bed for SSD research. With this motivation, we have designed and developed a simulation infrastructure. Here are the salient features and contributions of our work.

- The components of an SSD are can be classified as those belonging to the hardware and the software categories. The hardware component consists of a processing unit, memory, bus, and Flash chips. The software component (which executes on the processing unit) consists of a FTL. The price of the SSD depends on the hardware configuration in the SSD and the software running on the hardware, but there is a lack of test infrastructure to examine cost-effective hardware configurations and soft-

were algorithms in research environments outside those affiliated with manufacturers of SSDs. In this work, we provide an experimental test-bed to fill this void.

- The few efforts that have attempted to provide the simulation infrastructure [13], [3] lack desirable features, especially an object-oriented design. It is typically difficult to understand and enhance these simulators. Compared to other existing/evolving SSD simulators, FlashSim is entirely object-oriented. Our approach allows the developers to easily understand, use, and extend our simulator. Furthermore, our simulator has been integrated with the well-regarded and popular DiskSim simulator [7] and validated for behavioral similarity with real SSD devices.
- Energy consumption in SSDs is surprisingly higher than initially expected; energy consumption is approximately the same as mobile HDDs [17]. Thus, it is important to understand the causes of the energy consumption. We have analyzed the energy consumption in SSDs with our simulator by considering a simple energy model including various FTLs with real traces (Financial and TPC-H).

The rest of this paper is organized as follows: In Section 2, we present the basics of NAND Flash memory technology. We present the design of FlashSim and its implementation details in Section 3. We present the experimental results in Section 4. We discuss related work in Section 5. Finally, we summarize our work and discuss future direction in Section 6.

2. Background

Basics of NAND Flash Memory Technology. The most popular flash type for storage media is NAND flash memory due to higher density and lower cost than NOR flash. NAND flash provides three different operations: read, write, and erase. Each operation requires different operation time and granularity: Erase operations are performed at the granularity of a *block* that is composed of multiple *pages*. A page is the granularity at which reads and writes are performed. In addition to its data area, a page contains a small spare Out-of-Band area (OOB) which is used for storing a variety of information including: (i) Error Correction Code (ECC) information used to check data correctness, (ii) the logical page number corresponding to the data stored in the data area, and (iii) page state. Each page on flash can be in one of three different states: (i) *valid*, (ii) *invalid*, and (iii) *free/erased*. When no data has been written to a page, it is in the free/erased state. A write can be done only to a free page and changes its state to valid. An erase operation on an entire block of pages is required to revert the pages back to the free/erased state. Out-of-place updates result in certain written pages that are no longer valid. They are called invalid pages. Table 1 shows comparisons for different flash types in terms of access time and data unit size [20].

The Flash Translation Layer. The FTL is mainly composed of three software components (address translation, garbage collector, and wear-leveler), but the FTL is generally thought of as the address translation layer. The address translation layer that translates logical addresses from the file system

into physical addresses on flash devices helps in emulating flash as a normal block device; the layer performs out-of-place updates which in turn help to hide the erase operation in the flash memory. The mapping table is stored in a small, fast on-board SSD RAM. The garbage collector is in charge of collecting invalid pages to create free space in the flash memory. Since the lifetime of flash memory is limited by the number of erase operations on its cells (each memory cell typically has a lifetime of 10K-1M erase operations [6]), the wear-leveler elongates the lifetime of flash by maintaining the same level of wear for every block in the flash memory.

Flash Type	Data Unit Size			Access Time		
	Page (Bytes)	Block (Bytes)		Page READ (us)	Page WRITE (us)	Block ERASE (ms)
	Data	OOB				
Small Block	512	16	(16K+512)	41.75	226.75	2
Large Block	2048	64	(128K+4K)	130.9	405.9	2

Table 1: NAND Flash organization and access time comparison for Small-Block vs. Large-Block schemes [20].

State-of-The-Art FTL Scheme. FTLs can be implemented at different granularities of how large an address space a single entry in the mapping table captures. Page-based FTLs map the logical page number of the request sent to the device from the upper layers, such as file system, to any physical page on flash. Such translation requires a large mapping table to be stored in RAM. At the other extreme, a block-level FTL scheme only translates the logical block number into a physical block number; the logical page number offset within the block remains fixed, thus reducing the mapping table. To address the shortcomings of the above two extreme mapping schemes, researchers have come up with a variety of alternatives. Although many schemes have been proposed [5], [10], [15], [9], [16], they share one fundamental design principle. Each scheme is a *hybrid* between page-level and block-level schemes. The schemes logically partition their blocks into two groups - *Data Blocks* and *Log/Update Blocks*. Data blocks form the majority and are mapped using a block-level mapping technique, whereas the log blocks are mapped using a page-level mapping technique.

3. SSD Simulator Design

We have designed and implemented a SSD simulator that is based on the hardware diagram in Figure 1. The first version of our SSD simulator focused on software components (for instance, FTL schemes, garbage collection, and wear-leveling); we considered a simplified hardware model that simulated a single *Plane* with a simplified channel implementation.

Since this version of our simulator was limited by a simplified hardware model and not easy to extend due to a highly coupled implementation with DiskSim, we re-designed and re-implemented the simulator with an object-oriented approach. Our new simulator is entirely event-driven and written in a familiar language, C++; we achieve modularity, low coupling, and high cohesion. Our hardware-level diagram is shown in Figure 1.

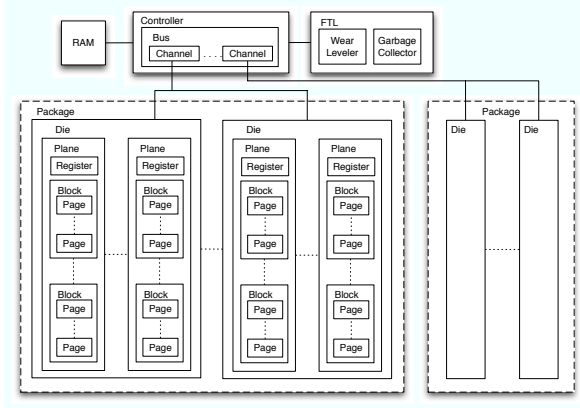


Figure 1: Hardware diagram for the SSD Simulator. Ellipses in between two of the same components indicate where more of the same components may be added. Only the full component breakdown of the left-most package is shown.

3.1. Object-Oriented Component Design

The simulator was written as a single-threaded program in C++ for simplicity. C++ could provide a comprehensible object-oriented scheme where each class instance represented a hardware or software component. The UML diagram of all C++ classes used by the SSD simulator is in [12]. FlashSim is integrated with Disksim’s C code. The classes in the SSD simulator for hardware and software components are as follows:

3.2. Hardware Component Design

- **SSD:** The SSD class serves to provide an interface to Disksim and provide a single class to instantiate in order to create the SSD simulator module. The SSD class creates event objects to wrap the Disksim *ioreq_event* structures and returns the event time to disksim.
- **Package:** The package class represents a group of flash dies that share a bus channel. The package class allocates its dies in its constructor and connects the dies to a bus channel. The package also facilitates addressing.
- **Die:** A die is a single flash chip organized into a set of planes. Dies are connected to bus channels, but individual planes contained in the die buffer bus transfers. In future development, the highest level at which merge operations may take place will be at the die level. The corresponding event object is updated with the merge delay time.
- **Plane:** Planes are comprised of blocks and provide a single page-sized register to buffer page data for bus transfers. The register is also used as a buffer for merge operations inside planes. The corresponding event object is updated with merge delays for merge operations and considers register delays.
- **Block:** A block is comprised of pages and is the smallest component that can be individually erased. When a block is erased, all pages in it are erased and can then be written

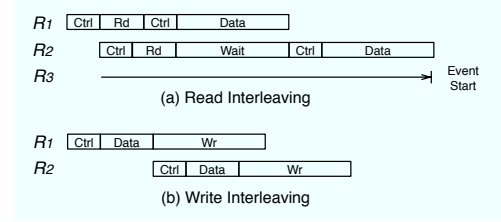


Figure 2: Interleaving for read/write requests

to again. The corresponding event object is updated with the erase delay time. A block can only be erased a finite number of times because of reliability constraints [4].

- **Page:** Each page maintains its state and updates event objects with the read and write delays of the given flash technology. Page states include free/empty after erasure, valid after a successful write, and invalid after being copied to a new location in a merge operation.
- **Controller:** The controller class receives event objects from the SSD and consults the FTL regarding how to handle each event. The controller sends the virtual data for events to the RAM for buffering before sending the event object to the bus.
- **RAM:** The RAM class calculates how long it takes to read or write data to itself. The RAM buffers virtual event data for the controller to send across the bus.
- **Bus:** The bus class has a number of channels that are each shared by all the dies in a package. The bus examines addresses in events and passes the event object on to the proper channel.
- **Channel:** Channels must schedule usage for events and update the event time values. Each channel keeps a scheduling table that keeps track of channel usage, and new events are scheduled at the next available free time slot after dependencies have been met. The scheduling table size is synonymous to queue length.

3.3. Software Component Design

- **Event:** First, the event class keeps track of its corresponding Disksim *ioreq_event* structure. Second, the event class holds methods and attributes to do all the record-keeping for the SSD simulator’s state, including SSD addresses. Simulator objects pass event class objects and update the event objects statistics.
- **Address:** Addresses are comprised of a separate field for each hardware address level from the package down to the page. We provide an address class instead of a *struct* to help make a clear interface to assign and validate addresses.
- **FTL:** The FTL provides address translation from logical addresses to physical addresses. It determines how to process events that involve many pages by producing a list of single-page events to be processed in-order by the controller. The FTL is responsible for taking advantage of hardware parallelism for performance. The FTL also has a wear leveler and garbage collector to facilitate its tasks.

- **Wear Leveler:** The wear leveler class helps spread the block erasures over all blocks in the SSD. The wear leveler is responsible for keeping as many blocks functional for as long as possible because blocks of pages can only be erased for reuse a finite number of times.
- **Garbage Collector:** The garbage collector is activated when a write request cannot be satisfied because the selected block is not writable or there is not enough free space in the selected block. The garbage collector seeks to merge partially-used blocks and free up blocks by erasing them. Any other algorithm for GC can also be simulated.

3.4. Bus Channel Interleaving

Figure 2 shows the interleaving of processing events for one bus channel. As per Figure 1, each bus channel connects to several flash dies that are grouped in a package. Each bus channel functions independently and in parallel; operations on different channels are not dependent on each other. The read interleaving for one bus channel is shown in Figure 2-(a). First, the control time signifies when the bus channel is locked for control signals that request a flash die to prepare data from a specific page. Next, the flash die processes the request for the data to be read. The bus channel is free to handle other requests at this time. Finally, the bus channel is locked for control signals that request the flash die to send data from a specific page and sending the data. The interesting part of this figure is the bus channel idle time period between the end of the control time for request two (R_2) and the beginning of the second control time period for request one (R_1). A control time period for request three cannot fit; request three (R_3) must be delayed until after request two finishes. The write for one bus channel is shown in Figure 2-(b). First, the bus channel must be locked for control signals to inform the proper flash die that it will receive data. Second, the bus remains locked to send the data. Finally, the flash die writes the data; the bus channel is free to handle other requests at this time. Since write requests only require one contiguous time block of bus channel time, write request happen in FIFO.

3.5. Event Flow

The SSD simulator is instantiated as a SSD object designed to accept *ioreq_event* structures from Disksim. Its functionality is described in detail in Algorithm 1. The SSD controller uses the FTL software module to create a list of events for a multi-page request. The controller issues each event in the list to the data hardware through corresponding bus channels. The bus channels handle the scheduling and interleaving of events for the controller; this simplifies our controller implementation. In Algorithm 2, events continue through the package and are handled starting at the die level; merge events can be handled inside flash dies or planes. Erase events are handled inside blocks, and read and write events are handled inside pages.

Input: Disksim's I/O Request Structure (*ioreq_event*)
Output: Device Service Time
foreach *ioreq_event* **do**
 begin SSD process *ioreq_event*
 wrap in *event* object;
 begin controller, FTL process *event*
 consult *wearleveler* and *garbagecollector*;
 create page-sized list of *event* objects;
 foreach *e* in *event_list* **do**
 begin SSD, bus, channel process *e*
 lock for next available transfer time;
 $e_{time} \leftarrow e_{time} + channel_delay$;
 end
 Package(*e*);
 end
 if $e_{type} = erase$ **then**
 update SSD wear stats;
 end
 begin inform bus, channel: *e* finished
 channel update scheduling table for event dependencies;
 end
 end
 end
end

Algorithm 1: SSD simulator functionality

Input: Event object (*e*)
Output: NULL
begin package, die process *e*
 /* Merge event *e* in die */
 if $e_{type} = merge$ and $e_{addr.plane} \neq e_{addr.merge.plane}$ **then**
 foreach valid page *v* in $e_{addr.block} x$ **do**
 foreach empty page *t* in $e_{addr.merge.block} y$ **do**
 $t \leftarrow v$;
 $v_{state} \leftarrow invalid$;
 $t_{state} \leftarrow valid$;
 end
 $e_{time} \leftarrow e_{time} + die_merge_delay$;
 end
 end
 /* Merge event *e* in plane */
 else plane process *e*
 $plane_{register} \leftarrow e_{data}$;
 if $e_{type} = merge$ **then**
 foreach valid page *v* in $e_{addr.block} x$ **do**
 foreach empty page *t* in $e_{addr.merge.block} y$ **do**
 $t \leftarrow v$;
 $v_{state} \leftarrow invalid$;
 $t_{state} \leftarrow valid$;
 end
 $e_{time} \leftarrow e_{time} + die_merge_delay$;
 end
 end
 /* $e_{type} = read$ or $write$ or $erase$ */
 else
 begin block process *e*
 if $e_{type} = erase$ **then**
 for each page in block *x* **do**
 $page_{state} \leftarrow empty$;
 end
 $e_{time} \leftarrow e_{time} + erase_delay$;
 update wear stats;
 end
 /* $e_{type} = read$ or $write$ */
 else page process *e*
 if $e_{type} = read$ **then**
 $e_{time} \leftarrow e_{time} + read_delay$;
 end
 else if $e_{type} = write$ **then**
 $e_{time} \leftarrow e_{time} + write_delay$;
 end
 end
 end
 if $e_{type} = erase$ **then**
 update plane, die, package wear stats;
 end
 end
 end

Algorithm 2: Package (event object) - SSD hardware functionality inside a package. This function is being called in Algorithm 1.

The SSD and package components are included in the call stack after consulting the bus channel because these components also keep track of wear statistics. Wear statistics stored in the SSD, package, die, plane, and block are updated every time an erase event occurs to keep a simple interface with lower algorithmic complexity for the FTL.

4. Experimental Results

We validated our simulator by comparing it to real SSDs for behavioral similarity; we compared the performance of different FTL schemes for realistic workload traces. We used the simplified version of the simulator that simulates a single *Plane* with a simplified channel implementation for various software implementations, such as the FTL, garbage collector, and wear-leveler. More thorough evaluation that also considers interleaving with parallelism effects is left for future work.

4.1. Evaluation Setup

The specifications available for commercial SSDs are insufficient for modeling them accurately. For example, the memory cache size for FTL mappings and the exact FTL scheme used are not disclosed. Hence, it is difficult to simulate these commercial devices. We made assumptions for flash devices as described in Table 2 and configured our simulator accordingly. Table 3 presents the salient features of our workloads.

Default simulation parameters		Real SSD Device	
Flash Type	Large Block	Real SSD1	Real SSD 2
Page (Data)	2KB	MSP-7000	FSD32GB25M
Page (OOB)	64B	MTron	Super Talent
Block	(128KB+4KB)	2.5 in	2.5 in
Page Read	(130.9 us, 4.72uJ)	4-way SLC	SLC
Page Write	(405.9 us, 38.04uJ)	Read: 120 MB/s	60 MB/s
Block Erase	(1.5 ms, 527.68uJ)	Write: 90 MB/s	45 MB/s
Interface	SATA		
GC	Yes		
Wear-leveling	Implicit/Explicit		
FTL Type	Page/DFTL		

Table 2: Simulation parameters and real SSD device observed specifications.

Workloads	Avg. Req. Size (KB)	Read (%)	Seq. (%)	Avg. Req. Inter-arrival Time (ms)	Simulated Time (sec)
Financial [18]	4.38	9.0	2.0	133.50	43,712
TPC-H [21]	12.82	95.0	18.0	155.56	37,058

Table 3: Enterprise-scale workload characteristics.

4.2. Validation of SSD Simulator

Using the parameters from Table 2, we validated our flash device simulator against commercial SSDs (MTron's SSD [1] and Super-Talent's SSD [2]) for *behavioral similarity*. For this purpose, we sent raw I/O requests to real SSDs and similar traces to our flash device simulator to measure

device performance. As shown in Figure 3, our simulator was able to capture the performance trends exhibited by the real SSDs. With increasing sequentiality of writes (Figure 3-(a)), the performance of real SSDs improved, and our flash simulator with various FTLs was able to provide similar characteristics. When examining reads (Figure 3-(b)), real SSDs showed much less variation; the same was observed with our simulator. With a high degree of randomness in writes (80% random in Figure 3-(c)), real SSDs demonstrated long-tailed response time distribution (due to larger GC overhead); our simulator exhibited a similar trend.

4.3. Evaluation

We conducted a comparison of performance and energy consumption according to different FTL schemes, including a page-based FTL, FAST [15], and DFTL [8]. We assumed the memory was just sufficient to hold the address translations for FAST. Since the actual memory size is not disclosed by device manufacturers, our estimate represents the minimum memory required for the functioning of a typical hybrid FTL. We allocated extra space (approximately 3% of the total active region [9]) for use as log-buffers by the hybrid FTL (FAST).

4.3.1. Performance Analysis. The Cumulative Distribution Function of the average system response time for different workloads is shown in Figure 4. DFTL is able to closely match the performance of the page-based FTL for the Financial trace. In comparison with the page-based FTL, DFTL reduces the total number of block erases as well as the extra page read/write operations by about 3 times. This results in improved device service times and shorter queuing delays; this improvement in turn improves the overall I/O system response time by about 78% as compared to FAST. For read-oriented workloads, DFTL incurs a larger additional address translation overhead, and its performance deviates from the page-based FTL. When considering TPC-H (in Figure 4(b)), however, FAST exhibits a *long tail* primarily because of the expensive full merges and the consequent high latencies seen by requests in the I/O driver queue. Hence, even though FAST services about 95% of the requests faster, it suffers from long latencies in the remaining requests, resulting in a higher average system response time than DFTL.

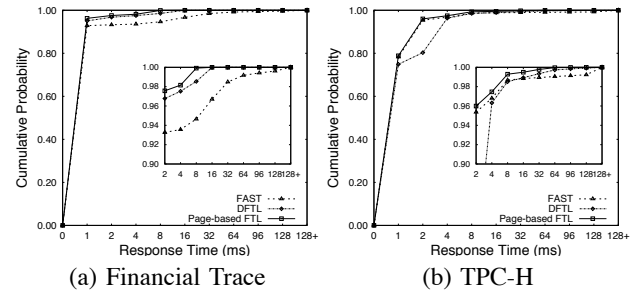


Figure 4: Cumulative Distribution Function of the average system response time for different FTL schemes.

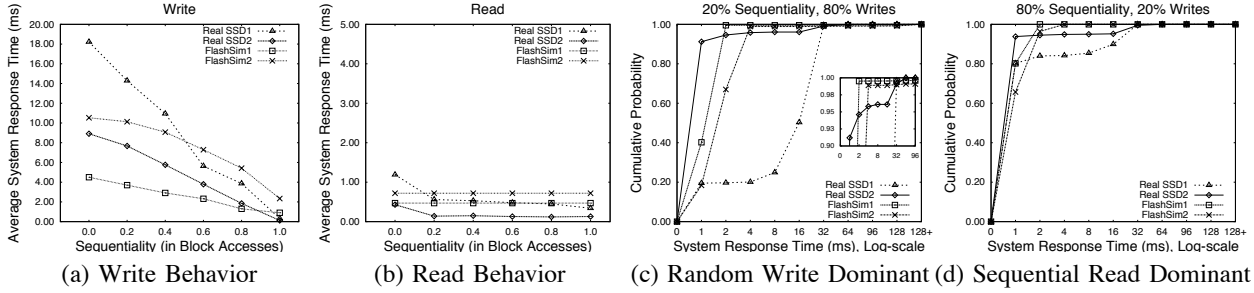


Figure 3: Validation of our SSD Simulator. Note that in the legends, Real SSD1, Real SSD2, FlashSim1, and FlashSim2 denote Mtron’s SSD, SuperTalent’s SSD, a SSD using a page-based FTL, and a SSD using DFTL.

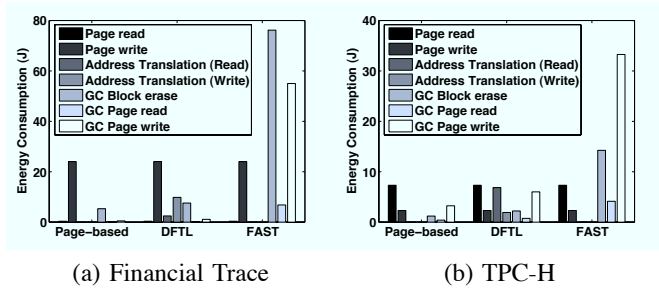


Figure 5: Energy consumption by different FTL schemes.

4.3.2. Analysis of Energy Consumption. Power consumption of the flash memory in the SSD may not be significant when compared to other components (CPU and Memory), but as shown in Table 2, erase operations consume significant power. Unlike individual read and write operations, erase operations have a greater impact on the overall SSD’s energy consumption, and the number of erase operations for a given workload varies according to the current FTL scheme. Figure 5 shows the energy consumption by operations for different FTL schemes in the Financial and TPC-H traces. The Financial trace is mostly random-write-dominant, while TPC-H is read-dominant (see Table 3). Thus, the energy consumption for the Financial trace is much higher than that for TPC-H due to the power consumed by GCs. DFTL requires additional page read and write operations due to mapping table entry misses in the memory, causing additional energy consumption in both traces. As expected, FAST FTL consumes significantly more energy than other FTL schemes due to more erase and write operations during GC.

In addition to power consumption by flash operations, the processor power consumption can be considerably high during GC. GC involves victim block searching overhead, which aims at finding the block with the least number of valid pages in order to reduce page copying overhead. Figure 6 shows the tradeoff between normalized average response time and the number of FTL search operations during GC for the Financial trace. Higher search operations decreases the response time while consuming more energy because (i) blocks with fewer valid pages require fewer

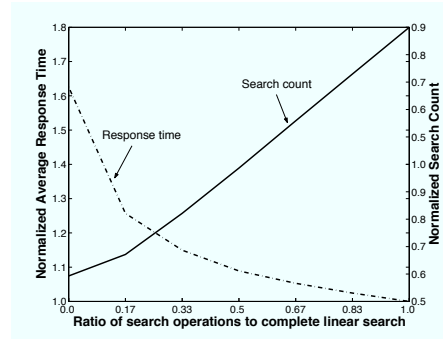


Figure 6: Tradeoff between performance and search operation energy consumption. This experiment has been conducted with DFTL for the Financial trace. We varied the number of search operations. Note that 0.0 on the X-axis means that the victim block is selected randomly without any search, and 1.0 means the victim block with the least number of invalid pages is selected after a complete linear search.

copy operations, and (ii) the search operations induce energy consumption by processor and system bus usage. Thus, the energy consumption during GC can be reduced by balancing fewer search operations with a greater number of copy operations. Fewer search operations will slightly increase response time because an incomplete search may select blocks with more valid pages that must be copied.

On-board RAM is another considerable factor in the power consumption in the SSD. Since the page-based FTL requires more memory as compared to the block-based FTL, the idle power consumption of the additional memory will be larger. FAST maintains block-level mapping for data regions and page-level mapping for log regions; the on-board RAM’s energy consumption is as close to that of the block-level FTL. DFTL requires the same memory as the block-level FTL; the idle power consumption is the same as that of the block-level FTL.

5. Related Work

Other research has been conducted to develop a simulator for NAND flash-based SSDs [3], [13]. Microsoft Research’s

simulator [3] is one of the first available SSD simulators; however, it is highly coupled with DiskSim. The strengths of their simulator include the implementation of parallelism effects across multiple channels and interleaving across different components within a single plane, but only a page-based FTL scheme is available. J. Lee et. al have developed a simple flash based SSD simulator [13]. This simulator is a stand-alone simulator that is limited by a single FTL scheme implementation, and they do not simulate I/O queuing effects.

Compared to the above simulators, our simulator has ability to simulate multiple FTL schemes, including page-based, block-based, FAST [15], and DFTL [8]. Our simulator is integrated with DiskSim to simulate queuing effects, and our simulator module can be instantiated multiple times within DiskSim. Our single-threaded, event-driven, object-oriented approach is comprehensible and modular to allow for future extensions. Furthermore, we have validated FlashSim against real SSD devices for behavioral similarity.

6. Summary and Future Work

We have developed a flexible and robust simulator for SSDs that features an object-oriented design. We have validated our simulator with real SSD devices by demonstrating behavioral similarity and compared performance results for various FTL schemes. We also have analyzed the impact of various FTL schemes on performance and power consumption in the SSD.

This project is a work in progress. Since the simulator has only been validated with a simple behavioral model for a single plane and simplified channel implementation, we will continue with more thorough validation methods that include bus channel interleaving effects. Caching and I/O scheduling effects will be added and examined. Since our simulator module can have multiple instances in DiskSim, we can simulate disk arrays that contain a combination of both SSDs and HDDs. In addition to performance simulation, our simulator is able to incorporate power models and other extensions. We plan to combine our thermal-performance simulator of disk drives [11] with our future work involving hybrid disk arrays that contain a combination of both SSDs and HDDs.

7. Download

Source-code is available for download from <http://csl.cse.psu.edu/hybridstore>.

Acknowledgments

We would like to thank Euseong Seo for his detailed comments which helped us improve the quality of this paper. This research was funded in part by NSF grants CCF-0811670, CNS-0720456, and a gift from Cisco, Inc.

References

[1] 2.5" MTron SATA Solid State Drive - MSP 7000. http://www.mtron.net/English/Product/ec_msp7000.asp.

- [2] 2.5" Super-Talent SATA Solid State Drive. <http://www.supertalent.com/products/ssd-commercial.php?type=SATA>.
- [3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *Proceedings of the USENIX Annual Technical Conference*, pages 57–70, June 2008.
- [4] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo. Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design. In *Proceedings of the 44th Annual Conference on Design Automation*, pages 212–217, New York, NY, USA, 2007. ACM.
- [5] T. Chung, D. Park, S. Park, D. Lee, S. Lee, and H. Song. System Software for Flash Memory: A Survey. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, pages 394–404, August 2006.
- [6] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. *ACM Computing Survey*, 37(2):138–163, June 2005.
- [7] G. Ganger, B. Worthington, and Y. Patt. *The DiskSim Simulation Environment Version 3.0 Reference Manual*.
- [8] A. Gupta, Y. Kim, and B. Ugaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 229–240, March 2009.
- [9] J. Kang, H. Jo, J. Kim, and J. Lee. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 161–170, October 2006.
- [10] J. Kim, J. Kim, S. Noh, S. Min, and Y. Cho. A Space-Efficient Flash Translation Layer for Compactflash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.
- [11] Y. Kim, S. Gurumurthi, and A. Sivasubramaniam. Understanding the Performance-Temperature Interactions in Disk I/O of Server Workloads. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2006.
- [12] Y. Kim, B. Tauras, A. Gupta, and B. Ugaonkar. FlashSim: A Simulator for NAND Flash-based Solid-State Drives. In *Technical Report CSE 09-008, Penn State University*, May 2009.
- [13] J. Lee, E. Byun, H. Park, J. Choi, D. Lee, and S. H. Noh. CPS-SIM: Configurable and accurate clock precision solid state drive simulator. In *Proceedings of the Annual ACM Symposium on Applied Computing (SAC)*, pages 318–325, March 2009.
- [14] S. Lee and B. Moon. Design of Flash-based DBMS: An In-Page Logging Approach. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 55–66, August 2007.
- [15] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song. A Log Buffer based Flash Translation Layer Using Fully Associative Sector Translation. *IEEE Transactions on Embedded Computing Systems*, 6(3):18, 2007.
- [16] S. Lee, D. Shin, Y. Kim, and J. Kim. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. In *Proceedings of the International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED2008)*, February 2008.
- [17] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating enterprise storage to ssds: Analysis of tradeoffs. In *Proceedings of the ACM European Conference on Computer Systems (Eurosys)*, pages 145–158, March 2009.
- [18] OLTP Trace from UMass Trace Repository. <http://traces.cs.umass.edu/index.php/Storage/Storage>.
- [19] S. Park, J. Park, J. Jeong, J. Kim, and S. Kim. A Mixed Flash Translation Layer Structure for SLC-MLC Combined Flash Memory System. In *Proceedings of the 1th International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED2008)*, 2008.
- [20] Technical Report (TN-29-07): Small-Block vs. Large-Block NAND Flash Devices. <http://www.micron.com/products/nand/technotes>.
- [21] J. Zhang, A. Sivasubramaniam, H. Franke, N. Gautam, Y. Zhang, and S. Nagar. Synthesizing Representative I/O Workloads for TPC-H. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2004.